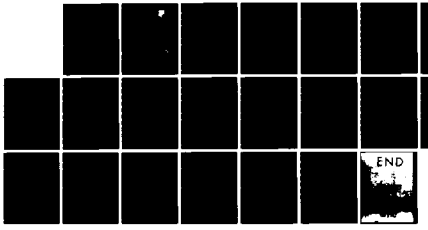
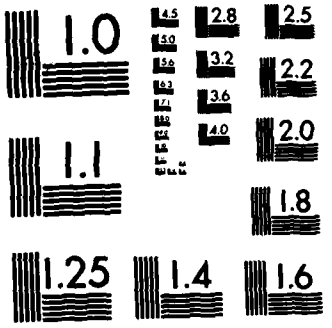
AD-A144 368 COMPILATION AND ENVIRONMENT OPTIMIZATIONS FOR LOGLISP (U) ROME AIR DEVELOPMENT CENTER GRIFFISS AFB NY R C SCHRAG JUL 84 RADC-TM-84-14 1/1

UNCLASSIFIED F/G 9/2 NL



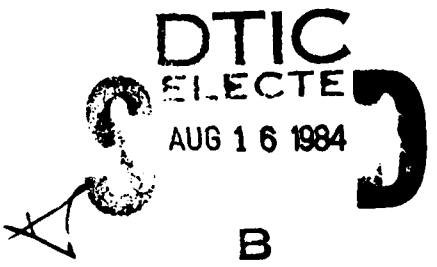


(13)

RADC-TM-84-14
In-House Report
~~June~~ July 1984

COMPILATION AND ENVIRONMENT OPTIMIZATIONS FOR LOGLISP

Robert C. Schrag



ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TM-84-14 has been reviewed and is approved for publication.

APPROVED: Samuel A. DiNitto Jr.

SAMUEL A. DINITTO JR.
Chief, Command & Control Software Technology Branch
Command & Control Division

APPROVED: David L Carlstrom

DAVID L. CARLSTROM, Colonel, USAF
Chief, Command & Control Division

FOR THE COMMANDER: Donald A Brantingham

DONALD A. BRANTINGHAM
Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A144368

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: N/A
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): RADC-TM-84-14	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: Rome Air Development Center; 6b. OFFICE SYMBOL (If applicable): COES	7a. NAME OF MONITORING ORGANIZATION: N/A
6c. ADDRESS (City, State and ZIP Code): Griffiss AFB NY 13441	7b. ADDRESS (City, State and ZIP Code):
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable): COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N/A
8c. ADDRESS (City, State and ZIP Code): Griffiss AFB NY 13441	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
62702F	5581	19	12

11. TITLE (Include Security Classification): COMPILATION AND ENVIRONMENT OPTIMIZATIONS FOR LOGLISP

12. PERSONAL AUTHOR(S): Robert C. Schrag

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
In-House	FROM Feb 83 TO Dec 84	July 1984	34

16. SUPPLEMENTARY NOTATION:

17. COSATI CODES		
FIELD	GROUP	SUB. GR.
9	2	14
9	2	15

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Programming languages, programming environments, LogLisp, Logic programming, compilation, Lisp, structure sharing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report documents investigation of technical issues involved in improving the execution efficiency of LogLisp through the employment of logic programming compilation. LogLisp is a hybrid language combining logic programming and Lisp developed by Syracuse University under contract to RADC. The version of LogLisp discussed here is known as V2M3.

The report consists of an Introduction followed by seven technical sections. The construction of a compiler for a programming language has the prerequisite that the virtual machine upon which its code will execute be completely specified. Logic programming compilers presently exist only for Prolog. Section 2 provides a concise model of Prolog interpretation. Section 3 describes commonly used Prolog space optimizations, and Section 6 Prolog compilation technology, as they apply to that model. These sections on Prolog characterize very tersely implementation concepts and techniques to establish a foundation for discourse of the current technical investigation.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Robert C. Schrag	(315) 330-2748	RADC (COES)

DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

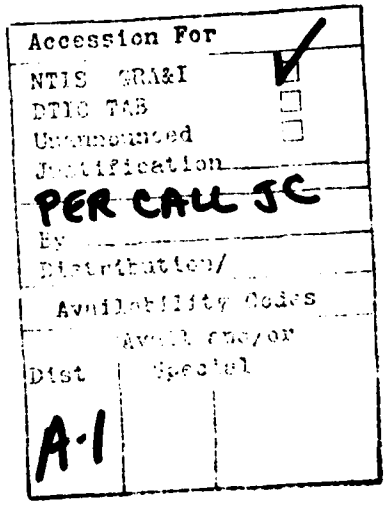
The present LogLisp interpreter is modeled in Section 4. The design decisions implemented in that interpreter to represent search paths and environments with structure sharing techniques and to fix certain arbitrary control assumptions (including ordering of clauses in a procedure and of calls in a clause body) are also assumed in Section 5, which offers possible space optimizations, and Section 7, which presents a scheme for LogLisp compilation. The Conclusion contains suggestions for future research and critical observations.

1. Introduction.

This report documents investigation of technical issues involved in improving the execution efficiency of LogLisp through the employment of logic programming compilation. LogLisp is a hybrid language combining logic programming and Lisp developed by Syracuse University under contract to RADC. The version of LogLisp discussed here is known as V2M3, and is described in [Robinson and Sibert 81a] and [Robinson and Sibert 81b].

The remainder of this report consists of seven sections. The construction of a compiler for a programming language has the prerequisite that the virtual machine upon which its code will execute be completely specified. Logic programming compilers presently exist only for Prolog. Section 2 provides a concise model of Prolog interpretation. Section 3 describes commonly used Prolog space optimizations, and Section 6 Prolog compilation technology, as they apply to that model. These sections on Prolog characterize very tersely implementation concepts and techniques described in [Bruynooghe 82], [Warren 77], and [Warren 80], to establish a foundation for discourse of the current technical investigation.

The present LogLisp interpreter is modeled in Section 4. The design decisions implemented in that interpreter to represent search paths and environments with the techniques of [Boyer and Moore 72] and to fix certain arbitrary control assumptions (including ordering of clauses in a procedure and of calls in a clause body) are also assumed in Section 5, which offers possible space optimizations, and Section 7, which presents a scheme for LogLisp compilation. The Conclusion contains suggestions for future research and critical observations.



2. Prolog Interpretation.

The vast majority of existing logic programming implementations are of Prolog—a purely backtracking, deterministic dialect. Prolog's rigid search strategy makes an implementation based on a contiguous area of memory which expands and contracts as a stack rather straightforward. It is easy to program a Prolog interpreter in a high-level language whose run-time stack can serve the same purpose, but this is less efficient than manipulating memory directly. The direct stack representation is also at the core of Warren's logic programming compilation techniques.

A Prolog stack frame corresponds to one application of resolution. It must, in general, include at least five fields of information. A pointer to the machine representation of the clause used and remaining clauses of the same procedure is required so that the next alternative clause to use upon failure is known. A similar pointer to the current call and its right-hand siblings is required so that the point at which to continue upon success is known. An environment for the variables of the clause the call is resolved with is required for the results of unification to be recorded. At least three types of control information must also be included in the frame: a pointer to the father stack frame containing the current call and its binding environment; a field indicating the most recent sibling, ancestor, or ancestor sibling frame with remaining alternative clauses to backtrack to; and a field for recording the locations of variable cells in previous stack frames which are written into as a result of this resolution and must be reset when this frame is backtracked through.

The algorithm which operates on the stack model begins by creating a new stack frame for the current call and copying the present frame's location into the father field. If this is the first call in the present frame, the backtrack field is copied from the father frame. For subsequent calls the backtrack field is set to the frame of the most recent non-determinate left-hand sibling (or one of its descendents) if it exists, otherwise as for a first call. The procedure corresponding to the call is retrieved and copied into the procedure field. Its first clause is examined and the required variables are allocated in the environment field. Unification is attempted, with results being recorded in this new and ancestor environment fields. Since frames are discarded on backtracking, it is required that references between variables be oriented from more recent toward less recent. Inefficient chains of reference are usually avoided by chasing variable bindings to their ultimate origins (dereferencing) before making assignments. Any ancestor variables that are instantiated as result of this unification are recorded in the reset field. If unification fails, the recorded

instantiations are undone, and an alternative clause of the procedure is tried. If no alternative clause exists, this and any intervening frames until that specified by the backtrack field are discarded, and execution resumes there with the selection of an alternative clause. If unification succeeds, the right-hand sibling of the present call in the father frame is executed. Subsequent answers to a query which succeeds once are obtained by resuming execution at the backtrack point of the successful frame.

3. Prolog Optimizations.

The essential stack model described above is somewhat wasteful of stack space in that frames are only reclaimed upon backtracking. This design cannot take advantage of the fact that a frame may be inherently, or may become through exhaustion of alternatives, deterministic, offering no further promise of new information.

A deterministic frame could be overwritten upon invocation of the sibling call, but for the possibility that that sibling shares a variable instantiated in that frame to a complex term, which itself contains variables. Overwriting the deterministic frame and its descendents would cause loss of the environment for the complex term's variables, creating a "dangling reference."

This dilemma is solved by creating a new, sister stack for the environments of variables which appear within complex terms to which ancestor variables are instantiated. This stack is popped only on backtracking (failure), allowing deterministic frames on the main stack and their ordinary variables to be overwritten, or "popped on success." (Subsequently this optimization is referred to as success popping.) The variables of complex terms are now preserved for ancestor frames, and are called global variables. The new stack is called the global stack. The main stack and its variables are referred to as local. It must now contain, in addition to the usual control information, a pointer to the corresponding frame in the global stack.

There is a further major opportunity to reclaim the stack space of deterministic frames. When the last call in a clause is reached and all of its siblings have terminated determinately (and hence been overwritten), retention of the father stack frame can contribute no promise of new information, and might be overwritten by the frame to be associated with the last call. (This is referred to subsequently as last call optimization.) Again it is necessary to preserve information. Variable bindings recorded in the father must be copied into a temporary location before that frame is overwritten so that they can be accessed during unification. The father's control information which specifying the next call to be executed upon success must also be saved, and is copied into the new frame. That is, the continuation information previously available in the call field is now maintained as an explicit list of calls, since the ancestor frames which contained it in the unoptimized model may now be overwritten.

While both these optimizations reduce run-time storage requirements, they reclaim storage which would in any case be released upon backtracking. The optimization of determinate last calls (also

called tail recursion optimization when the last call is recursive on the procedure) has the very attractive practical advantage of allowing "perpetual processes" to be implemented in application programs [Warren 82]. This optimization also is impossible without the separation of variables onto local and global stacks that allows determinate frames to be overwritten.

Whenever frames are overwritten, the possibility exists that the only pointers to (or need for) information on the global stack may be deleted, creating (or exposing) garbage. In order to support processes of significant duration, especially under last call optimization (where the eventuality of backtracking to reclaim space is not guaranteed), a garbage collector must be implemented for the global stack. Warren [-- 77] points out that such a garbage collector is quite complex, involving, beyond the classic marking and sweeping, compaction of the global stack with remapping of pointers from the local stack into it.

4. LogLisp Interpretation.

LogLisp is a hybrid language, combining Lisp reduction/evaluation with inference in a heuristic, non-backtracking implementation of logic programming in Lisp. The Lisp semantics of LogLisp includes all of standard Lisp plus reduction of Lisp forms used in Logic--the logic programming semantics of LogLisp. This section primarily addresses the unique search algorithm of Logic. Because multiple search paths are explored simultaneously, it is impossible for LogLisp to use Prolog-style stack techniques. Instead, active nodes of the search space (each corresponding to a search path) are kept on a heap. Instead of a contiguous area of memory, linked list data structures which can be easily shared across the contexts of different search paths are used. These design decisions follow [Boyer and Moore 72].

A heap node consists of a segment list, or lifetime continuation containing all outstanding unsolved predications of the node, and an environment containing the bindings for variables instantiated on this path. The environment is represented in the heap as a list of association lists, each with variable bindings for the predications resulting from resolution with a particular clause, distinguished by a unique environment index. Association list entries at an index use the bound variable name as association key, and show the binding expression and the index at which to view in turn any of its variables. Each segment in the segment list consists of predications forming some tail of the body of a clause resolved with on this path, prefaced by the index at which their variables are viewed. (This is a simplified description. A node can also contain a structure associated with the COND special form of Logic, which is omitted here without loss of generality.)

The LogLisp execution cycle begins by removing a node (of low heuristic cost) from the heap and spreading its environment list in a global environment array so that association lists can be accessed directly by their indices, without the cost of traversing a linked list. The first predication in the segment list of the selected node is Lisp-reduced, and then its resolvent nodes are computed from each clause in the selected procedure. The reduction machinery actually implements conditional evaluation of variable-containing Lisp forms, reducing them only as much as is allowed by their states of variable instantiation. If reduction succeeds in completely evaluating a call and its result is not NIL, then the next predication in the segment list is processed in the same way. If the result is NIL, then that call is considered to fail. Resolution with the clauses of a called procedure is attempted after reduction. The variables of a resolving clause are viewed at the next available index. The current environment is extended by pushing

variable binding entries resulting from unification onto the environment array at the viewing index. If a variable in the head of a clause unifies with a term in the selected predication, the binding entry extends the newly created index. If the binding must be made in the other direction, then the entry extends the index of the bound variable. Complete dereferencing of variables applies in either case.

Segment list tails, environment list tails, and individual association lists or their tails are shared among heap nodes whenever possible. Other than reserving an index for the association list, no allocation of space for a clause's variables is made, and no extension of association lists is performed until a variable is bound. Pushing onto a list does not alter tail structure, so this policy allows association list tails to be shared across contexts. When unification with a clause is successful, the entries of the extended environment array are gathered into list representation, sharing the largest common environment list tail with the current environment. Sharing of association lists across contexts occurs when unextended association lists are gathered. Sharing of segment list tails occurs when the new segment resulting from a non-unit resolving clause is pushed onto the segment list.

A resolvent node containing the new segment list and gathered environment is entered onto the heap. The environment array is restored to its unextended state by spreading the current environment again before attempting unification with another clause of the same procedure. When all resolvent nodes (search path extensions) of this node have been generated, another (low-cost) node is removed from the heap. The process of expanding heap nodes into their resolvents continues until the heap is empty or a satisfactory number of solutions has been found.

LogLisp Interpretation.

Figure 1 is provided to clarify the LogLisp execution cycle. It lists extremely compact Lisp (pseudo-) code to represent a stylized interpreter.

```
(deduce (lambda (continuation)
   (prog (call environment solved node procedure)
    in: (setq solved nil)
        (enter-heap-node (cons continuation nil))
   run: (cond ((null (setq node (remove-heap-node)))
               (go out:)))
        (setq continuation (car node))
        (spread (setq environment (cdr node)))
        (setq call (car continuation))
        (setq continuation (cdr continuation))
 simpl: (setq call (reduce call))
        (cond ((evaluable call)
               (cond ((not (null result-of-call))
                      (cond (continuation (setq call (car continuation))
                                          (setq continuation (cdr continuation))
                                          (go simpl:))
                            (t (setq solved (cons environment solved))
                               (go run:)))
                      (t nil)))   {call is reduced}
              (t (setq procedure (name call))
                 (prog (clause)
                  next: (setq clause (car procedure))
                        (spread environment)
                        (cond ((unify (head clause) call)
                               (cond ((setq continuation
                                            (append (body clause)
                                                    continuation))
                                      (enter-heap-node
                                             (cons continuation
                                                   (gather))))
                                     (t (setq solved
                                              (cons (gather)
                                                    solved))))))
                        (cond ((null (setq procedure (cdr procedure)))
                               (go run:)))
                        (go next:))))
   out: (return solved))))
```

Figure 1

LogLisp Interpretation.

The details of segment indexing have been omitted in Figure 1, and the segment list is represented by a homogeneous continuation. The query calls are passed to deduce as such a continuation. The variable solved contains a list of solution environments from which query variable bindings can be obtained.

LogLisp Interpretation.

Figure 2 shows a determinate LogLisp program and a trace of the contents of each successive node as it is entered onto and removed from the heap in execution.

```
                                                        {(comments)}
 (|- (FOO x) <- (BAR x) & (TOT y) & (COMP z))           {four
 (|- (BAR (ZIP a)))                                      procedures
 (|- (TOT (ZAP b)))                                      in the
 (|- (COMP THIS))                                        program}

(THE v (FOO v))                                         {query to run}
(ZIP a:2)                                               {answer}


Cycle 0:
segment list    ((0 (FOO v)))                           {initial call,
environment     ((0))                                    index 0}

Cycle 1:
segment list    ((1 (BAR x)(TOT y)(COMP z)))            {new segment,
environment     ((1 (x 0 . v))                           new index,
                 (0))                                    x:1 <- v:0}

Cycle 2:
segment list    ((1 (TOT y)(COMP z)))                   {advance over
environment     ((2)                                     segment.  new
                 (1 (x 0 . v))                           index for var's
                 (0 (v 2 . (ZIP a))))                    of BAR clause}

Cycle 3:
segment list    ((1 (COMP z))                           {advance again.
environment     ((3)                                     another new
                 (2)                                     index}
                 (1 (y 3 . (ZAP b))(x 0 . v))
                 (0 (v 2 . (ZIP a))))

Cycle 4:
segment list    ()                                      {query solved!
environment     ((4)                                     read off
                 (3)                                     answer from v:0}
                 (2)
                 (1 (z 4 . THIS)(y 3 . (ZAP b))(x 0 . v))
                 (0 (v 2 . (ZIP a))))
```

Figure 2

LogLisp Interpretation.

In Cycle 0 the query call is packaged into a segment list with variables to be viewed at environment index 0, which initially has no binding entries. This call is removed from the segment list in Cycle 1 and replaced with a new segment corresponding to the body of the FOO clause. Variable x at index 1 (x:1) is bound to variable v at index 0 (v:0--still unbound). In Cycle 2 the BAR call is processed. Its argument variable, x, is dereferenced, leading to the binding of v:0 to (ZIP a:2). Environment index 2 is dedicated to bindings of BAR clause variables. In Cycle 3 the environment association list at index 1 is extended to indicate the binding of y:1 to (ZAP b:3). In Cycle 4 it is extended again, and the segment list becomes empty, representing solution of the query, found as the binding of v:0.

5. LogLisp Optimizations.

This section addresses the import of the conventional logic programming space reclamations, success popping and last call optimization, in the context of LogLisp's unconventional search machinery. These optimizations exist so that the Prolog stack machine can avail itself of the space occupied by frames which are determinate in the sense that they offer no further promise of new information. The absence of backtracking in LogLisp means that all nodes are determinate. This is certainly evidenced by LogLisp's discarding of any heap node after its resolvents have been generated.

When a local stack frame is overwritten in Prolog, storage for both the responsible call and environment for the resolving clause's variables are released. In the case of last call optimization, the continuation is copied into the new frame. In LogLisp's current implementation, however, there is no distinction between local and global variables, and only one environment list for each node is maintained. Therefore, while the parent call is always removed from the continuation in constructing the descendent node, all bindings for either type of variable in the association list at the environment index created by that call is always retained. Since the environment array which is used for fast indexing is of finite length, the maintaining of indices containing only useless bindings has the unfortunate effect of causing space in this array for new rule application to be exhausted prematurely. The most severe disadvantage of this limitation is that it prevents users from programming perpetual processes.

Ignoring for the moment the problem of how to store global information, the perspective that all nodes are determinate leads one to the conclusion that success popping and last call optimization might always be performed on the current environment. In terms of LogLisp structures, this means that a segment's last call could always reuse the segment's environment index, and that earlier calls in a segment could always use the index one greater than the segment's environment index. (Last call optimization requires provision of a temporary location for the index to be overwritten, and an answer location for answer template variables which must never be overwritten.) This would lead to situation in which any node would always have equal numbers of segments and environment entries (except when a unit rule clause had just been used). This sort of scheme would lead to very short environment lists, and less overhead in spreading and gathering during context switching.

To cope with global information, an array parallel to the environment array, a usage array, could be used to indicate whether a particular index would be needed for global information. A flag would

be set in this array if unification established a binding from an earlier index (of an earlier variable to a complex term containing variables) into the corresponding index. The set flag would signify that the index is used for global bindings and cannot be overwritten. This environment utilization requires that a list of free environment indices be maintained for each heap node. Success popping and last call optimization are performed as in the naive scheme above, except that when an index cannot be overwritten because its usage flag is set, the next index on the free list is chosen instead.

This scheme is very similar to environment utilization under space optimizations in Prolog. Flagged indices can be thought of as constituting the global environment, and unflagged indices the local environment. The number indices in the local environment equals the number of active segments (continuations), as in Prolog. The fact that local variable bindings also exist in "global" indices is unimportant, since an association list will accommodate an arbitrary number of variable bindings. The index is necessary for the implementation of structure-sharing across deduction contexts, and to avoid problems of variable name conflict. This scheme would be complete with the provision of garbage collection for global indices to clean up after last call optimization's abandonments.

A "mark and sweep" garbage collector would mark only indices. (Restructuring of their contents is prohibited by structure-sharing across deduction contexts.) It would proceed from all local indices, marking any global indices their variables point to. From this point, at least two approaches are possible. All of the bindings in a global index could be similarly chased at the gross index level, or more care could be taken to chase only the specific variables which were referenced, possibly freeing more indices. Gross chasing might be adequate in the average case.

The fact that "normal" unification algorithms, like LogLisp's, cannot create circular ("infinite") data structures means that a "reference count" implementation of garbage collection is also feasible. This would require another array parallel to the environment array--the keeping array--used by unification to record, when a variable in a corresponding earlier index is instantiated to a complex term containing variables, the later index referred to. The usage array must now hold for later indices their reference counts—showing how many earlier indices want to "keep" them.

Now garbage collection is done incrementally within the inference process. Success popping and last call optimization are performed as before, but before overwriting any index, the reference count of each entry it has kept is decremented (once for each appearance on the

keeping list of the index--duplicates are possible and meaningful). If any reference count so decremented reaches zero, its index is returned to the free list, with its own kept indices processed similarly in turn. This scheme is attractive in that it has the effect of "fine" garbage collection without requiring specific variable traversal after initial recording, even though indices are traversed at a gross level.

A "heap trace" of the program listed in Figure 2 showing the mechanics of last call optimization, success popping, and reference count garbage collection is provided as Figure 3.

```
LogLisp Optimizations.

Cycle 0:
segment list    ((0 (FOO v)))          {last call in segment:  optimize}
environment     ((0))
temporary        ---                   {nothing being overwritten}
answer          ()                     {no answer variables bound yet}
keeping         ((0))                  {no global
usage           ((0 0))                       variables yet}
free list       (1 2 3 ...)            {only index 0 in use}

Cycle 1:
segment list    ((0 (BAR x)(TOT Y)(COMP z)))    {success pop}
environment     ((0 (x answer . v)))             {index 0 reused}
temporary       ()                        {previous contents of index 0}
answer          ()
keeping         ((0))
usage           ((0 0))
free list       (1 2 3 ...)               {index 1 available}

Cycle 2:
segment list    ((0 (TOT y)(COMP z)))          {success pop, but...}
environment     ((1)(0 (x answer . v)))        {0 + 1 = index 1}
temporary        ---
answer          (v 1 . (ZIP a))                {answer variable bound}
keeping         ((1)(0)(answer (1)))           {x:answer
usage           ((1 1)(0 0))                        <- (ZIP a:1)}
free list       (2 3 4 ...)                    {...index 1 not available}

Cycle 3:
segment list    ((0 (COMP z)))                 {optimize last call}
environment     ((2)(1)(0 (y 2 . (ZAP b))
                          (x answer . v)))
temporary        ---
answer          (v 1 . (ZIP a))
keeping         ((2 1)(1 1)(0))                {y:0
usage           ((2)(1)(0 (2))(answer (1)))         <- (ZAP b:2)}
free list       (3 4 5 ...)

Cycle 4:
segment list    ()
environment     ((1)(0))
temporary       ((y 2 . (ZAP b))(x answer . v))
answer          (v 1 . (ZIP a))
keeping         ((1 1)(0))
usage           ((1)(0)(answer (1)))
free list       (2 3 4 ...)                    {index 2 again free}
```

Figure 3

LogLisp Optimizations.

In Cycle 0 index 0 holds the packaged query call and null environment, as in Figure 2. Index 0 entries also exist for the keeping and usage arrays, showing that no index 0 variables have been bound to later complex terms and that no index 0 complex terms (containing global variables) have been bound by earlier variables, respectively. There is no entry in the temporary location because last call optimization is not in effect. No answer variables have been instantiated and only index 0 is in use. Since there is only one call in the current segment, last call optimization is performed on this node, and in Cycle 1 index 0 is reused for the new segment. The temporary location holds the previous contents of the overwritten index (which was empty) for access during unification. Now there is more than one call in the current segment, so that success popping can be performed on this node. The index one greater than current index 0, index 1, is available, and is used for the variables of the BAR clause in Cycle 2. Dereferencing of x:0 leads to the binding of v:answer to (ZIP a:1), noted in the answer location. This binding establishes global variables, causing the reference count in index 1 of the usage array to be incremented and index 1 to be placed on the list at the answer index of the keeping array. There is still more than one call in the current segment, indicating success popping, but now index 1 is guarded from being overwritten by its non-zero reference count. Cycle 3 uses index 2 for the variables of the TOT clause. Another global variable binding is recorded in the coupled usage and keeping arrays. A single call remains in the current segment, indicating last call optimization. When index 0 is overwritten in Cycle 4 the reference count for index 2 (noted on the list at index 0 of the keeping array) is decremented to 0. All unguarded indices beyond overwritten index 0 (just index 2) are now returned to the free list.

Both of these schemes for environent optimization suffer new overhead due to new node-specific data structures. The keeping array must be extended, gathered, and spread in the same sense as the environment array. The usage array can contain integers (or flags) rather than lists and need not be extended, but must be gathered and spread. The free list is merely pushed and popped, and is unaffected by context switching.

Whatever sort of garbage collection is chosen, it will be less complicated than in the Prolog, stack-based case. Expensive compaction and remapping of indices is unnecessary because each holds exactly one pointer (to its association list) in the LogLisp, linked list-base case. The free list serves the analogous function here.

6. Prolog Compilation.

The stack model that has been described for Prolog interpretation is compatible with compiled execution of Prolog programs. In interpretive execution, procedures of the program are represented declaratively by data structures, and the search algorithm is implemented in an interpreter which performs operations, including frame initialization, unification, backtracking, and optimizations on the stack. In compiled execution, the procedures of the program are represented procedurally by executable code in which the search algorithm and its stack-manipulating operations have been embedded. A Prolog compiler transforms Prolog code to machine instructions for which a specific stack model serves as the virtual machine. The usage of the stack frame fields under compiled execution is the same as interpretively, except that contents of the call and procedure fields must be considered to be instructions. (In this discussion, compiled Prolog code is presented in a stylized way that highlights just the essential features of that described in [Warren 77] and [Warren 80].)

Prolog's deterministic search algorithm is embedded by the sequencing of instructions in compiled code. A procedure compiles into a sequence of instructions to try each member clause in turn, and is considered to succeed if any of its clauses succeed. A clause compiles into: matching instructions for unification, corresponding to the head of the clause; a sequence of call instructions to compiled procedures, corresponding to the body; and an exit instruction to transfer control to the parent goal's continuation. A clause is considered to succeed if its matching and all of its call instructions succeed.

Prolog Compilation.

Figure 4 is provided to illustrate the structure of compiled Prolog code.

```
try(firstclause);

    .
    .
    .

trylast(lastclause);
```

(a) procedure code structure

```
match;
call(firstprocedure);

    .
    .
    .

call(lastprocedure);
exit;
```

(b) clause code structure

Figure 4

A new procedure is entered (Figure 4(a)) upon resolution, and a new frame is pushed onto the stack. The continuation and father fields are filled in. As each member clause is tried, the procedure field is updated, the environment is allocated, and control passes to the executable code for the individual clause (Figure 4(b)). The clause's unification instructions dereference arguments of the call (using the father environment) and record the effects of unification of head variables, including making appropriate entries in the reset field. The call instructions call new procedures which establish subsequent stack frames. Finally, the exit instruction transfers control to the parent goal's continuation if it exists, and otherwise to the backtrack point.

While the transformation of procedures into independently executable code contributes somewhat to the efficiency advantage of compiled code, the greatest contribution is provided by the generation of matching instructions for the heads of clauses. These instructions speed the general unificiation algorithm by specializing it for particular arguments. The compiler generates such specialized

unification instructions for each argument in a clause head.

The technique of creating efficient specializations of algorithms with respect to a known, particular subset of its arguments is called partial evaluation [Emanuelson 82]. Such specialization can be performed ad hoc, or by an automated partial evaluation system (cf. [Kahn 83]) operating on a program in a designated language and given arguments. Through this technique, much unnecessary testing, dead code, and function-calling overhead can be eliminated.

Warren [— 77] (p50-51) lists some compiler optimizations which correspond to partial evaluation of unification. These include: binding first occurrences of head variables directly, without matching; omitting instructions for top-level (void) variables with just one occurrence; avoiding recursive calls to unification by associating nesting level information with head symbols; and overwriting cells for variables occurring only in the head but more than once, after the head has been matched.

Unification is further dispatched in Warren's virtual Prolog machine via the representation of code for which no instructions are generated (—unification in calls, or beyond a certain nesting level in heads) by literals, which associate type information with the individual symbols of a predication.

Other optimizations described by Warren include: delaying allocation of variable cells until successful unification has been determined; the use of indexing to narrow the search for matching clauses, share unification work among similar clause heads, and help detect determinism in procedures; and local/global variable categorization to enable stack optimizations.

Because it cannot be determined at compile-time whether a complex term will be bound to an ancestor variable or merely have its arguments matched, Warren's compiler generally classifies all variables occurring in complex terms as global, but also provides a mechanism for declaring whether an argument is to be used for input or output. These mode declarations restrict the use of procedures, but offer further opportunities for partial evaluation of unification. A complex term declared as input need have no space reserved for a molecule. A complex term declared as output need have no executable instructions generated for its arguments.

While it is difficult to sort out the relative benefit of any specific one of these optimizations, Warren [— 77] estimates that together they produce a 15- to 20-fold improvement in computation time efficiency, on average.

7. LogLisp Compilation.

The breadth-first character of LogLisp's search algorithm prohibits its being neatly embedded in executable code. Called procedures must generate resolvent nodes and suspend, rather than going on to call other procedures directly. The virtual machine for compiled LogLisp code must consist of the heap and environment and associated arrays, in contrast to the simple stacks of Prolog.

The primary decision to be made in determining the format of compiled code is where partial evaluation should be performed. Clause heads always unify against the calls of clause bodies, so it only makes sense to partially evaluate unification with respect to one or the other—not both. (Otherwise there would be ambiguity as to which atomic formula were to act upon the other.) Since unification with the head of a clause precedes unification with its body, there is more opportunity for efficiency enhancement here, by applying Warren's technique for direct binding on a variable's first occurrence. Calls, being subject to instantiation before execution, are fluid and more difficult to compile.

A Logic compiler can retain the Prolog compiler practice of generating for each procedure a sequence of instructions for resolution with its individual clauses. Reduction of a call must take place before the clauses are tried. The practice of generating matching instructions for clause heads can also be retained to achieve the primary benefit of logic programming compilation—partial evaluation. The code for an individual clause must spread the current environment and execute the clause's matching instructions. If matching is successful, it must then gather the environment and enter it together with the new segment list as a node on the heap. The final instruction in the procedure must be to remove a node from the heap and call the appropriate procedure. Environment optimizations can be performed at this time.

LogLisp Compilation.

Figure 5 is provided to suggest a possible structure for compiled Logic code.

```
(this-logic-procedure (lambda (call continuation environment)
   (progn (spread environment)
      (setq call (reduce call))
      (cond ((evaluable call)
             (cond ((not (null result-of-call))
                    (cond (continuation
                           (eval (list (name (car continuation))
                                       (car continuation)
                                       (cdr continuation)
                                       environment)))
                          (t (setq solved (cons environment solved))))
                    (t nil)))
            (t

               .
               .
               .

               {instructions for individual clauses}

               .
               .
               .

               (eval (remove-heap-node)))))))

               (a) procedure code structure

(progn (spread environment)
   (cond ((apply this-clause-matching-instructions call)
          (cond ((setq continuation (append this-clause-body
                                            continuation))
                 (enter-heap-node (list (name (car continuation))
                                        (car continuation)
                                        (cdr continuation)
                                        (gather))))
                (t (setq solved (cons (gather) solved))))
          (t nil))))

                 (b) clause code structure
```

Figure 5

LogLisp Compilation.

Figure 5(a) indicates Lisp-reduction and simplification of calls being performed before resolution with individual clauses is attempted. Figure 5(b) indicates execution of matching instructions generated by partial evaluation of unification with respect to the head of a particular clause, and the packaging of an executable procedure call for the resolvent node, if any. The last instruction in the procedure code structure invokes such a packaged call selected from the heap. Note the correspondance with Figure 1.

Instructions which implement partial evaluation of unification specifically for LogLisp must be developed. Some of Warren's Prolog techniques will be applicable. The first has already been mentioned. Detection of void variables is straightforward, and the omission of instructions to unify with them (except to advance past their mates) is applicable. Avoidance of recursive calls to unification should be left to a more complete investigation of partial evaluation in light of the mechanism chosen for preserving global information, but it may be desirable, if user-specifiable indexing on any of a procedure's terms or subterms is implemented, to generate instructions sufficient to expedite unification at the specified level. Overwriting individual variable cells loses significance in the case of LogLisp, since only the index is valuable, and structure sharing prohibits rearrangements within it. Because LogLisp supports the use of variable tails (...as opposed to null tails. Consider the resolution of a head "(P x y z)" against a call "(P . u)." This LogLisp mechanism effectively allows predicates and functors of variable arity.) permitting any tail of a Logic predication or term to be bound to an ancestor variable, it may be desirable to maintain for this eventuality purely "literal" copies of predications for which instructions have been generated.

Current LogLisp already uses the literal #VAR# for rapid identification of variables, though it is more expensive than in the Prolog case in which literals are essentially tagged, since the reliance on Lisp list structures requires it to occupy an extra cons cell. LogLisp also uses the less expensive literal # for void variables. Integers and atoms are well distinguished by the underlying Lisp system, so that literals for these are unnecessary. Skeleton literals in Prolog are used similarly to Lisp cons cells, and are likewise unnecessary in LogLisp. With or without the proposed environment optimizations, separate literals distinguishing local and global variables are unnecessary.

LogLisp already employs full secondary indexing, but it stores each assertion separately, and does not collapse unification instructions for similar clause heads. The proposed format for compiled code is compatible with Warren's indexing scheme, since it too represents

procedures by sequences of clause instructions. Warren's scheme calls for alternating specific and general sections of indexed code, corresponding to heads respectively with and without variable arguments, in order to reflect the program control which user-ordering of clauses defines. LogLisp always considers data before rules, so only one specifc and one general section will be needed for each procedure. In the rare case that the procedure name is the same as that of one of Logic's special resolution rules, that semantics must be accounted for in an intervening section.

In the context of the proposed environment optimizations, mode declarations could be used to eliminate the need to check for variables in a complex term to be bound to an ancestor variable to determine whether its index is to be kept. Arguments declared as input need never have the index kept; arguments declared as output must always have it kept. The variable-occurrence analysis requires traversing the term's structure until a variable literal is found, and is computationally expensive, so that logic programming systems usually avoid it whenever possible. Warren's Prolog compiler succeeds in that this traversal is performed once, at compile-time, but the existence of variable tails makes global variable identification in LogLisp impossible in the general case. A workable alternative to run-time variable-occurrence analysis in LogLisp might lie in the use of occurrence literals throughout the list structure of a clause. A tagged (virtual) Lisp architecture would generally serve to make the representation of Logic code by literals painlessly efficient. Then an occurrence literal could be simply an available flag in a cons cell used only in the list structure of predications, set if the arguments to CONS contained variables. Another alternative which would eliminate the need for analysis in some cases is the optional declaration of the ability of procedures to have their tails bound.

The generation of instructions implementing the partial evaluation of reduction might offer another opportunity to significantly enhance the efficiency of LogLisp execution. It has been tendered that unification instructions produce greatest benefit in clause heads. Reduction, on the other hand, is never performed on clause heads, but only on outstanding predications which derive from clause bodies. To allow head instructions for unification to continue to operate on these predications seems to require that any reduction instructions (to be executed only at the time a predication is selected for resolution) be treated by unification as (perhaps non-operative) literals.

Reduction instructions for Lisp system functions might be created by partially evaluating current LogLisp machinery which implements reduction for those functions with respect to expected arguments. Without declarations, it will be impossible to generate reduction

instructions for Lisp functions other than system functions, because LogLisp does not require a Logic identifier's Lisp definition (if any) to be available at the time of assertion. With identifier-type declarations, a hypothetical automated system might be used to generate reduction machinery implementing conditional evaluation, which could then be partially evaluated to produce reduction instructions for tractable Lisp functions. In any case, reduction instructions cannot be guaranteed to be maximally specific because of the vulnerability of calls to instantiation before their execution. In the absence of complete reduction instructions, variable occurrence literals could be used to at least determine to what extent an expression needed to be reduced or shown (reproduced as an instantiated copy for Lisp evaluation).

The speed-up resulting from these compilation techniques is likely to be less than the 15- to 20-fold achieved by Warren, considering the fact that LogLisp already includes some of the optimizations that contribute to that figure. A more conservative estimate might be a 5- to 10-fold improvement in processing speed. This estimate must be tempered against the fact that compilation is only successful when enough particularity can be determined at the time of assertion. LogLisp's rich and flexible semantics make this determination a more difficult proposition.

8. Conclusion.

It must be emphasized that the proposed optimization and compilation techniques for LogLisp are now no more than paper conceptualizations. Any practical value they may possess must be learned from their actual implementation. Their compatibility with the forthcoming V3M1 version of LogLisp, which performs modified backtracking, should also be examined. If an alternative multiple context management technique (such as "hash table windows," described in [Borgwardt 84]) is chosen over the structure sharing strategy currently employed, then the applicability of the proposed techniques will be limited. LogLisp cannot achieve the full benefit of compilation without the inclusion of reduction instructions. The automatic generation of reduction machinery for user-defined Lisp functions is an unexplored technical area that might deserve investigation.

Heuristic search is at once one of LogLisp's most distiguishing features and one of the greatest obstacles to its efficient implementation. The utility of this feature must be convincingly demonstrated. (Prolog is sometimes criticized for its reliance on "impure" deterministic control features, but these seem to have been embraced by the user community.) Conditional evaluation of Lisp forms also incurs high overhead, but much of this might be compiled away. It may be worthwhile to consider building a pure backtracking, deterministic LogLisp compiler, if the price of heuristic search is judged too high. Such an endeavor must still be weighed against the benefits of a more standard logic programming/Lisp interface, such as that found in Lisp Machine Prolog [Kahn 82].

References

[Borgwardt 84] P. Borgwardt. "Parallel Prolog Using Stack Segments on Shared-memory Multiprocessors," in the Proceedings of the 1984 International Symposium on Logic Programming, IEEE Computer Society Press.

[Boyer and Moore 72] R.S. Boyer and J.S. Moore. "The Sharing of Structure in Theorem-proving Programs," in B. Meltzer and D. Michie (eds), Machine Intelligence VII, John Wiley.

[Bruynooghe 82] M. Bruynooghe. "The Memory Management of Prolog Implementations," in K.L. Clark and S-A. Tarnlund (eds), Logic Programming, Academic Press.

[Emanuelson and Haraldsson 80] P. Emanuelson and A. Haraldsson. "On Compiling Embedded Languages in Lisp," Conference Record of the 1980 Lisp Conference, Stanford University.

[Emanuelson 82] P. Emanuelson. "From Abstract Model to Efficient Compilation of Patterns," Linkoping University Research Report LITH-MAT-82-03.

[Kahn 82] K.M. Kahn. "Unique Features of Lisp Machine Prolog," UPMAIL Technical Report 14, Uppsala University.

[Kahn 83] K.M. Kahn. "A Partial Evaluator of Lisp Written in Prolog," UPMAIL Technical Report 1983-02-17, Uppsala University.

[Komorowski 82] H.J. Komorowski. "A Prototype Prolog Compiler," Proceedings of the 6th Software Engineering Conference, Tokyo.

[Robinson and Sibert 81a] J.A. Robinson and E.E. Sibert. The LogLisp User's Manual, unpublished interim technical report, Syracuse University.

[Robinson and Sibert 81b] J.A. Robinson and E.E. Sibert. LogLisp Implementation Notes, unpublished interim technical report, Syracuse University.

[Warren 77] D.H.D. Warren. "Implementing Prolog—Compiling Predicate Logic Programs, v1," D.A.I. Research Report No. 39, University of Edinburgh.

References

[Warren 80] D.H.D. Warren. "An Improved Prolog Implementation which Optimizes Tail Recursion," in S-A. Tarnlund, (ed) Proceedings of the Logic Programming Workshop, Debrecen, Hungary.

[Warren 82] D.H.D. Warren. "Perpetual Processes--an Unexploited Prolog Technique," Logic Programming Newsletter, number 3, Universidade Nova de Lisboa.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

END

FILMED

DTIC